3 Message authentication codes

This chapter covers

- Message authentication codes (MACs)
- The security properties and the pitfalls of MACs
- The widely adopted standards for MACs

Mix a hash function with a secret key and you obtain something called a *message authentication code* (MAC), a cryptographic primitive to protect the integrity of data. The addition of a secret key is the foundation behind any type of security: without keys there can be no confidentiality, and there can be no authentication. While hash functions can provide authentication or integrity for arbitrary data, they do that thanks to an additional trusted channel that cannot be tampered with. In this chapter, you will see how a MAC can be used to create such a trusted channel and what else it can do as well.

NOTE For this chapter, you'll need to have read chapter 2 on hash functions.

3.1 Stateless cookies, a motivating example for MACs

Let's picture the following scenario: you are a web page. You're bright, full of colors, and above all, you're proud of serving a community of loyal users. To interact with you, visitors must first log in by sending you their credentials, which you must

then validate. If the credentials match the ones that were used when the user first signed up, then you have successfully *authenticated* the user.

Of course, a web browsing experience is composed not just of one, but of many requests. To avoid having the user re-authenticate with every request, you can make their browser store the user credentials and resend them automatically within each request. Browsers have a feature just for that—*cookies*! Cookies are not just for credentials. They can store anything you want the user to send you within each of their requests.

While this naive approach works well, usually you don't want to store sensitive information like user passwords in cleartext in the browser. Instead, a session cookie most often carries a random string, generated right after a user logs in. The web server stores the random string in a temporary database under a user's nickname. If the browser publishes the session cookie somehow, no information about the user's password is leaked (although it can be used to impersonate the user). The web server also has the possibility to kill the session by deleting the cookie on their side, which is nice.

There is nothing wrong with this approach, but in some cases, it might not scale well. If you have many servers, it could be annoying to have all the servers share the association between your users and the random strings. Instead, you could store more information on the browser side. Let's see how we can do this.

Naively, you can have the cookie contain a username instead of a random string, but this is obviously an issue, as I can now impersonate any user by manually modifying the username contained in the cookie. Perhaps the hash functions you learned about in chapter 2 can help us. Take a few minutes to think of a way hash functions can prevent a user from tampering with their own cookies.

A second naive approach could be to store not only a username, but a digest of that username as well in a cookie. You can use a hash function like SHA-3 to hash the username. I illustrate this in figure 3.1. Do you think this can work?

There's a big problem with this approach. Remember, the hash function is a public algorithm and can be recomputed on new data by a malicious user. If you do not trust

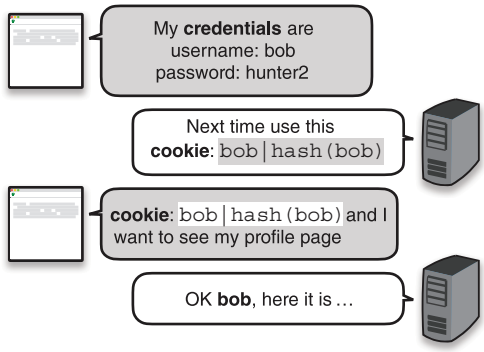


Figure 3.1 To authenticate the requests for a browser, a web server asks the browser to store a username and a hash of that username, sending this information in every subsequent request.

the origin of a hash, it does not provide data integrity! Indeed, figure 3.2 shows that if a malicious user modifies the username in their cookie, they can also simply recompute the digest part of the cookie.

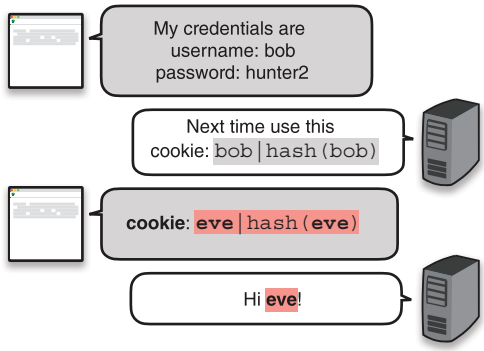


Figure 3.2 A malicious user can modify the information contained in their cookies. If a cookie contains a username and a hash, both can be modified to impersonate a different user.

Still, using a hash is not a foolish idea. What else can we do? Turns out that there is a similar primitive to the hash function, a MAC, that will do exactly what we need.

A *MAC* is a secret key algorithm that takes an input, like a hash function, but it also takes a secret key (who saw that coming?) It then produces a unique output called an *authentication tag*. This process is deterministic; given the same secret key and the same message, a MAC produces the same authentication tag. I illustrate this in figure 3.3.

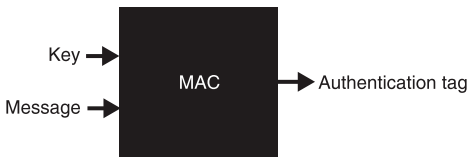


Figure 3.3 The interface of a message authentication code (MAC). The algorithm takes a secret key and a message, and deterministically produces a unique authentication tag. Without the key, it should be impossible to reproduce that authentication tag.

To make sure a user can't tamper with their cookie, let's now make use of this new primitive. When the user logs in for the first time, you produce an authentication tag from your secret key and their username and have them store their username and the authentication tag in a *cookie*. Because they don't know the secret key, they won't be able to forge a valid authentication tag for a different username.

To validate their cookie, you do the same: produce an authentication tag from your secret key and the username contained in the cookie and check if it matches the authentication tag contained in the cookie. If it matches, it must have come from you, as you were the only one who could have produced a valid authentication tag (under your secret key). I illustrate this in figure 3.4.

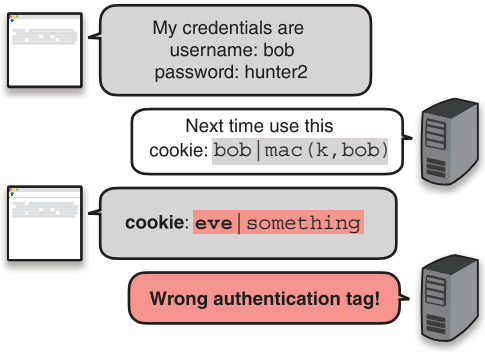


Figure 3.4 A malicious user tampers with his cookie but cannot forge a valid authentication tag for the new cookie. Subsequently, the web page cannot verify the authenticity and integrity of the cookie and, thus, discards the request.

A MAC is like a private hash function that only you can compute because you know the key. In a sense, you can personalize a hash function with a key. The relationship with hash functions doesn't stop there. You will see later in this chapter that MACs are often built from hash functions. Next, let's see a different example using real code.

3.2 An example in code

So far, you were the only one using a MAC. Let's increase the number of participants and use that as a motivation to write some code to see how MACs are used in practice. Imagine that you want to communicate with someone else, and you do not care about other people reading your messages. What you really care about, though, is the integrity of the messages: they must not be modified! A solution is to have both you and your correspondent use the same secret key with a MAC to protect the integrity of your communications.

For this example, we'll use one of the most popular MAC functions—*hash-based message authentication code* (HMAC)—with the Rust programming language. HMAC is a message authentication code that uses a hash function at its core. It is compatible with different hash functions, but it is mostly used in conjunction with SHA-2. As the

following listing shows, the sending part simply takes a key and a message and returns an authentication tag.

Listing 3.1 Sending an authenticated message in Rust

```
use sha2::Sha256;
use hmac::{Hmac, Mac, NewMac};

fn send_message(key: &[u8], message: &[u8]) -> Vec<u8> {
    let mut mac = Hmac::<Sha256>::new(key.into());

    mac.update(message);

    mac.finalize().into_bytes().to_vec()
}
```

Instantiates HMAC with a secret key and the SHA-256 hash function

Buffers more input for HMAC

Returns the authentication tag

On the other side, the process is similar. After receiving both the message and the authentication tag, your friend can generate their own tag with the same secret key and then compare those. Similarly to encryption, both sides need to share the same secret key to make this work. The following listing shows how this works.

Listing 3.2 Receiving an authenticated message in Rust

```
use sha2::Sha256;
use hmac::{Hmac, Mac, NewMac};

fn receive_message(key: &[u8], message: &[u8],
  authentication_tag: &[u8]) -> bool {
    let mut mac = Hmac::<Sha256>::new(key);
    mac.update(message);

    mac.verify(&authentication_tag).is_ok()
}
```

The receiver needs to recreate the authentication tag from the same key and message.

Checks if the reproduced authentication tag matches the received one

Note that this protocol is not perfect: it allows replays. If a message and its authentication tag are replayed at a later point in time, they will still be authentic, but you'll have no way of detecting that it is an older message being resent to you. Later in this chapter, I'll tell you about a solution. Now that you know what a MAC can be used for, I'll talk about some of the "gotchas" of MACs in the next section.

3.3 Security properties of a MAC

MACs, like all cryptographic primitives, have their oddities and pitfalls. Before going any further, I will provide a few explanations on what security properties MACs provide and how to use them correctly. You will learn (in this order) that

- MACs are resistant against forgery of authentication tags.
- An authentication tag needs to be of a minimum length to be secure.
- Messages can be replayed if authenticated naively.
- Verifying an authentication tag is prone to bugs.

3.3.1 Forgery of authentication tag

The general security goal of a MAC is to prevent *authentication tag forgery* on a new message. This means that without knowledge of the secret key, k, one cannot compute the authentication tag $t = MAC(k, m)$ on messages m of their choice. This sounds fair, right? We can't compute a function if we're missing an argument.

MACs provide much more assurance than that, however. Real-world applications often let attackers obtain authentication tags on some constrained messages. For example, this was the case in our introduction scenario, where a user could obtain almost arbitrary authentication tags by registering with an available nickname. Hence, MACs have to be secure even against these more powerful attackers. A MAC usually comes with a proof that even if an attacker can ask you to produce the authentication tags for a large number of arbitrary messages, the attacker should still not be able to forge an authentication tag on a never-seen-before message by themselves.

> **NOTE** One could wonder how proving such an extreme property is useful. If the attacker can directly request authentication tags on arbitrary messages, then what is there left to protect? But this is how security proofs work in cryptography: they take the most powerful attacker and show that even then, the attacker is hopeless. In practice, the attacker is usually less powerful and, thus, we have confidence that if a powerful attacker can't do something bad, a less powerful one has even less recourse.

As such, you should be protected against such forgeries *as long as the secret key used with the MAC stays secret*. This implies that the secret key has to be random enough (more on that in chapter 8) and large enough (usually 16 bytes). Furthermore, a MAC is vulnerable to the same type of ambiguous attack we saw in chapter 2. If you are trying to authenticate structures, make sure to serialize them before authenticating them with a MAC; otherwise, forgery might be trivial.

3.3.2 Lengths of authentication tag

Another possible attack against usage of MACs are *collisions*. Remember, finding a collision for a hash function means finding two different inputs X and Y such that $HASH(X) = HASH(Y)$. We can extend this definition to MACs by defining a collision when $MAC(k, X) = MAC(k, Y)$ for inputs X and Y.

As we learned in chapter 2 with the birthday bound, collisions can be found with high probability if the output length of our algorithm is small. For example, with MACs, an attacker who has access to a service producing 64-bit authentication tags can find a collision with high probability by requesting a much lower number (2^{32}) of tags. Such a collision is rarely exploitable in practice, but there exist some scenarios where collision resistance matters. For this reason, we want an authentication tag size that would limit such attacks. In general, 128-bit authentication tags are used as they provide enough resistance.

> *[requesting 2^{64} authentication tags] would take 250,000 years in a continuous 1Gbps link, and without changing the secret key K during all this time.*
>
> —RFC 2104 ("HMAC: Keyed-Hashing for Message Authentication," 1997)

Using a 128-bit authentication tag might appear counterintuitive because we want 256-bit outputs for hash functions. But hash functions are public algorithms that one can compute *offline*, which allows an attacker to optimize and parallelize an attack heavily. With a keyed function like a MAC, an attacker cannot efficiently optimize the attack offline and is forced to directly request authentication tags from you, which usually makes the attack much slower. A 128-bit authentication tag requires 2^{64} *online* queries from the attacker in order to have a 50% chance to find collisions, which is deemed large enough. Nonetheless, one might still want to increase an authentication tag to 256-bit, which is possible as well.

3.3.3 Replay attacks

One thing I still haven't mentioned are *replay attacks*. Let's see a scenario that is vulnerable to such attacks. Imagine that Alice and Bob communicate in the open using an insecure connection. In order to protect the messages from tampering, they append each of their messages with an authentication tag. More specifically, they both use two different secret keys to protect different sides of the connection (as per best practice). I illustrate this in figure 3.5.

Figure 3.5 Two users sharing two keys, `k1` and `k2`, exchange messages along with authentication tags. These tags are computed from `k1` or `k2`, depending on the direction of the messages. A malicious observer replays one of the messages to the user.

In this scenario, nothing prevents a malicious observer from replaying one of the messages to its recipient. A protocol relying on a MAC must be aware of this and build protections against this. One way is to add an incrementing counter to the input of the MAC as shown in figure 3.6.

In practice, counters are often a fixed 64-bit length. This allows one to send 2^{64} messages before filling up the counter (and risking it to wrap around and repeat itself).

Figure 3.6 Two users sharing two keys, `k1` and `k2`, exchange messages along with authentication tags. These tags are computed from `k1` or `k2`, depending on the direction of the messages. A malicious observer replays one of the messages to the user. Because the victim has incremented his counter, the tag will be computed over `2, fine and you?` and will not match the tag sent by the attacker. This allows the victim to successfully reject the replayed message.

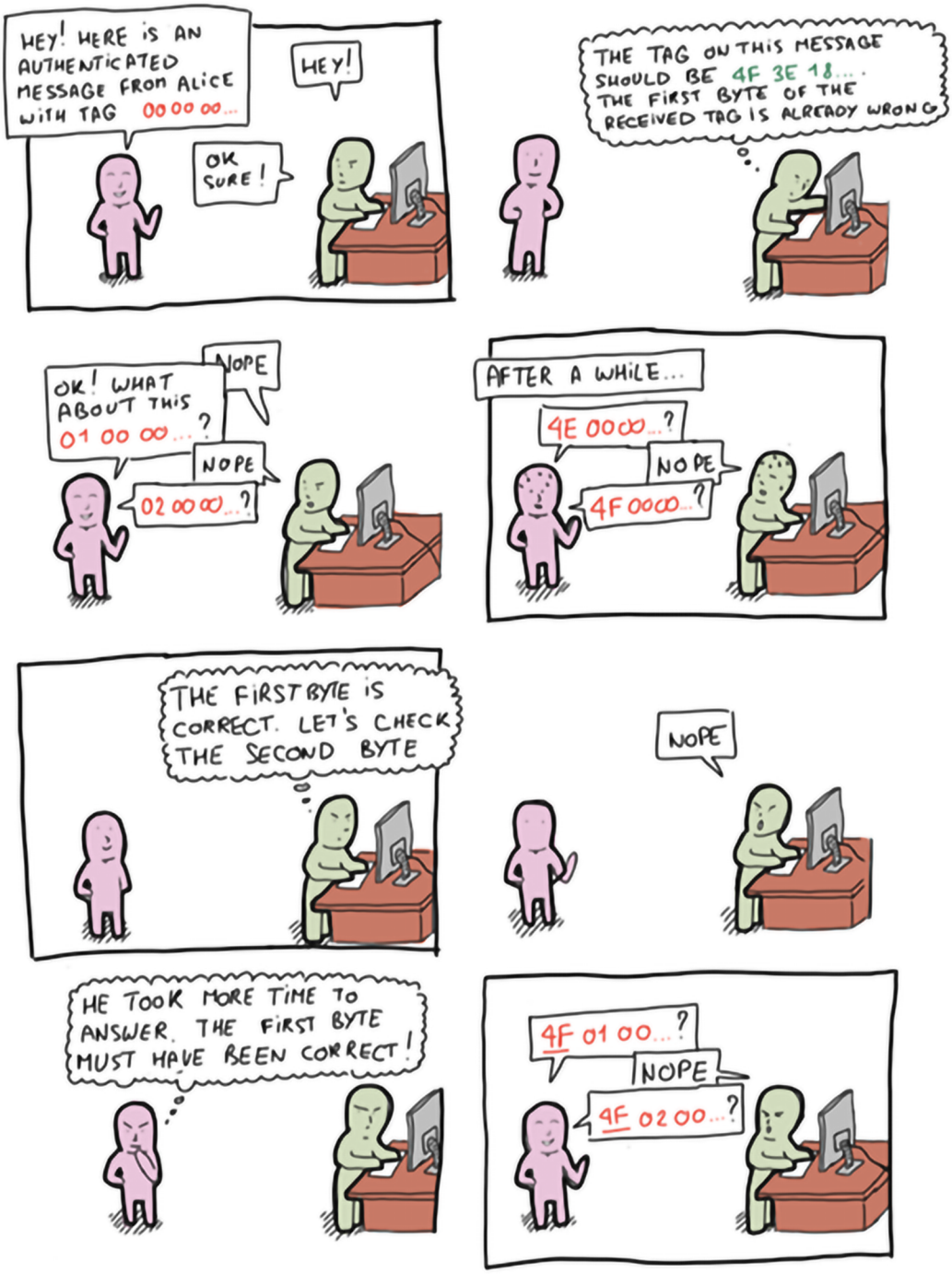
Of course, if the shared secret is rotated frequently (meaning that after X messages, participants agree to use a new shared secret), then the size of the counter can be reduced and reset to 0 after a key rotation. (You should convince yourself that reusing the same counter with two different keys is OK.) Again, counters are *never variable-length* because of ambiguous attacks.

> **Exercise**
> Can you figure out how a variable-length counter could possibly allow an attacker to forge an authentication tag?

3.3.4 Verifying authentication tags in constant time

This last gotcha is dear to me as I found this vulnerability many times in applications I audited. When verifying an authentication tag, the comparison between the received authentication tag and the one you compute must be done in *constant time*. This means the comparison should always take the same time, assuming the received one is of the correct size. If the time it takes to compare the two authentication tags is not constant time, it is probably because it returns the moment the two tags differ. This usually gives enough information to enable attacks that can recreate byte by byte a valid authentication tag by measuring how long it takes for the verification to finish. I explain this in the following comic strip. We call these types of attacks *timing attacks*.

Fortunately for us, cryptographic libraries implementing MACs also provide convenient functions to verify an authentication tag in constant time. If you’re wondering how this is done, listing 3.3 shows how Golang implements an authentication tag comparison in constant time code.


HEY! HERE IS AN AUTHENTICATED MESSAGE FROM ALICE WITH TAG 00 00 00...
HEY!
OK SURE!
THE TAG ON THIS MESSAGE SHOULD BE 4F 3E 18.... THE FIRST BYTE OF THE RECEIVED TAG IS ALREADY WRONG
OK! WHAT ABOUT THIS 01 00 00...?
NOPE
02 00 00...?
NOPE
AFTER A WHILE...
4E 00 00...?
NOPE
4F 00 00...?
THE FIRST BYTE IS CORRECT. LET'S CHECK THE SECOND BYTE
NOPE
HE TOOK MORE TIME TO ANSWER. THE FIRST BYTE MUST HAVE BEEN CORRECT!
4F 01 00...?
NOPE
4F 02 00...?

Listing 3.3 Constant time comparison in Golang

```
for i := 0; i < len(x); i++ {
    v |= x[i] ^ y[i]
}
```

The trick is that no branch is ever taken. How this works exactly is left as an exercise for the reader.

3.4 MAC in the real world

Now that I have introduced what MACs are and what security properties they provide, let's take a look at how people use them in real settings. The following sections address this.

3.4.1 Message authentication

MACs are used in many places to ensure that the communications between two machines or two users are not tampered with. This is necessary in both cases where communications are in cleartext and where communications are encrypted. I have already explained how this happens when communications are transmitted in cleartext, and in chapter 4, I will explain how this is done when communications are encrypted.

3.4.2 Deriving keys

One particularity of MACs is that they are often designed to produce bytes that look random (like hash functions). You can use this property to implement a single key to generate random numbers or to produce more keys. In chapter 8 on secrets and randomness, I will introduce the HMAC-based key derivation function (HKDF) that does exactly this by using HMAC, one of the MAC algorithms we will talk about in this chapter.

The pseudorandom function (PRF)

Imagine the set of all functions that take a variable-length input and produce a random output of a fixed size. If we could pick a function at random from this set and use it as a MAC (without a key), it would be swell. We would just have to agree on which function (kind of like agreeing on a key). Unfortunately, we can't have such a set as it is way too large, but we can emulate picking such a random function by designing something close enough: we call such constructions *pseudorandom functions (PRFs)*. HMAC and most practical MACs are such constructions. They are randomized by a key argument instead. Choosing a different key is like picking a random function.

Exercise

Caution: not all MACs are PRFs. Can you see why?

3.4.3 Integrity of cookies

To track your users' browser sessions, you can send them a random string (associated to their metadata) or send them the metadata directly, attached with an authentication tag so that they cannot modify it. This is what I explained in the introduction example.

3.4.4 Hash tables

Programming languages usually expose data structures called *hash tables* (also called hashmaps, dictionaries, associated arrays, and so on) that make use of noncryptographic hash functions. If a service exposes this data structure in such a way where the input of the noncryptographic hash function can be controlled by attackers, this can lead to *denial of service* (DoS) *attacks*, meaning that an attacker can render the service unusable. To avoid this, the noncryptographic hash function is usually randomized at the start of the program.

Many major applications use a MAC with a random key in place of the noncryptographic hash function. This is the case for many programming languages (like Rust, Python, and Ruby), or for major applications (like the Linux kernel). They all make use of *SipHash*, a poorly-named MAC optimized for short authentication tags, with a random key generated at the start of the program.

3.5 Message authentication codes (MACs) in practice

You learned that MACs are cryptographic algorithms that can be used between one or more parties in order to protect the integrity and the authenticity of information. As widely used MACs also exhibit good randomness, MACs are also often used to produce random numbers deterministically in different types of algorithms (for example, the time-based one-time password [TOTP] algorithm that you will learn in chapter 11). In this section, we will look at two standardized MAC algorithms that one can use nowadays—HMAC and KMAC.

3.5.1 HMAC, a hash-based MAC

The most widely used MAC is HMAC (for *hash-based MAC*), invented in 1996 by M. Bellare, R. Canetti, and H. Krawczyk, and specified in RFC 2104, FIPS Publication 198, and ANSI X9.71. HMAC, like its name indicates, is a way to use hash functions with a key. Using a hash function to build MACs is a popular concept as hash functions have widely available implementations, are fast in software, and also benefit from hardware support on most systems. Remember that I mentioned in chapter 2 that SHA-2 should not be used directly to hash secrets due to *length-extension attacks* (more on that at the end of this chapter). How does one figure out how to transform a hash function into a keyed function? This is what HMAC solves for us. Under the hood, HMAC follows these steps, which I illustrate visually in figure 3.7:

1. It first creates two keys from the main key: $k1 = k \oplus ipad$ and $k2 = k \oplus opad$, where *ipad* (inner padding) and *opad* (outer padding) are constants, and $\oplus$ is the symbol for the XOR operation.

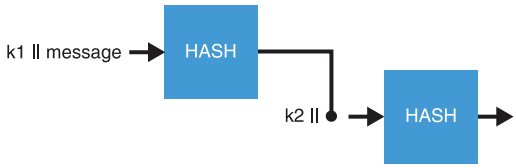


Figure 3.7 HMAC works by hashing the concatenation (||) of a key, `k1`, and the input message, and then by hashing the concatenation of a key, `k2`, with the output of the first operation. `k1` and `k2` are both deterministically derived from a secret key, `k`.

2. It then concatenates a key, `k1`, with the message and hashes it.
3. The result is concatenated with a key, `k2`, and hashed one more time.
4. This produces the final authentication tag.

Because HMAC is customizable, the size of its authentication tag is dictated by the hash function used. For example, HMAC-SHA256 makes use of SHA-256 and produces an authentication tag of 256 bits, HMAC-SHA512 produces an authentication tag of 512 bits, and so on.

> **WARNING** While one can truncate the output of HMAC to reduce its size, an authentication tag should be at minimum 128 bits as we talked about earlier. This is not always respected, and some applications will go as low as 64 bits due to explicitly handling a limited amount of queries. There are tradeoffs with this approach, and once again, it is important to read the fine print before doing something nonstandard.

HMAC was constructed this way in order to facilitate proofs. In several papers, HMAC is proven to be secure against forgeries as long as the hash function underneath holds some good properties, which all cryptographically secure hash functions should. Due to this, we can use HMAC in combination with a large number of hash functions. Today, HMAC is mostly used with SHA-2.

3.5.2 KMAC, a MAC based on cSHAKE

As SHA-3 is not vulnerable to length-extension attacks (this was actually a requirement for the SHA-3 competition), it makes little sense to use SHA-3 with HMAC instead of something like `SHA-3-256(key || message)` that would work well in practice. This is exactly what *KMAC* does.

KMAC makes use of cSHAKE, the customizable version of the SHAKE extendable output function (XOF) that you saw in chapter 2. KMAC unambiguously encodes the MAC key, the input, and the requested output length (KMAC is some sort of extendable output MAC) and gives this to cSHAKE as an input to absorb (see figure 3.8). KMAC also uses "KMAC" as function name (to customize cSHAKE) and can, in addition, take a user-defined customization string.

Interestingly, because KMAC also absorbs the requested output length, several calls with different output lengths provide totally different results, which is rarely the case for XOFs in general. This makes KMAC quite a versatile function in practice.

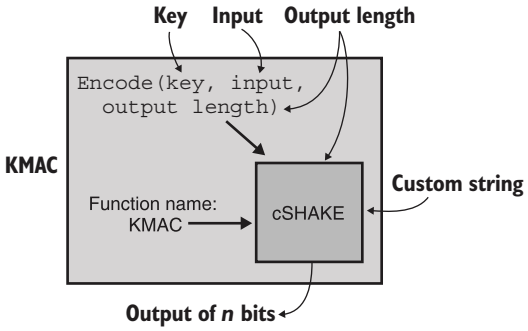


Figure 3.8 KMAC is simply a wrapper around cSHAKE. To use a key, it encodes (in a unambiguous way) the key, the input, and the output length as the input to cSHAKE.

3.6 SHA-2 and length-extension attacks

We have mentioned several times that one shouldn't hash secrets with SHA-2 as it is not resistant to *length-extension attacks*. In this section, we aim to provide a simple explanation of this attack.

Let's go back to our introduction scenario, to the step where we attempted to simply use SHA-2 in order to protect the integrity of the cookie. Remember that it was not good enough as the user can tamper with the cookie (for example, by adding an `admin=true` field) and recompute the hash over the cookie. Indeed, SHA-2 is a public function and nothing prevents the user from doing this. Figure 3.9 illustrates this.

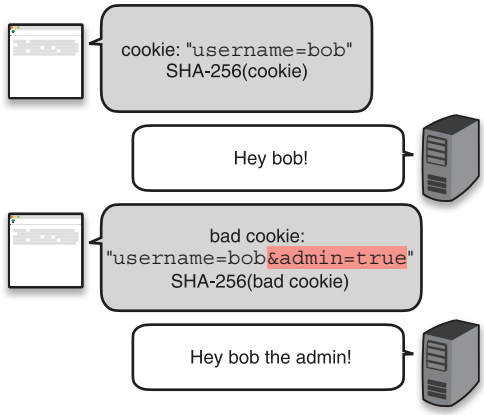


Figure 3.9 A web page sends a cookie followed by a hash of that cookie to a user. The user is then required to send the cookie to authenticate themselves in every subsequent request. Unfortunately, a malicious user can tamper with the cookie and recompute the hash, breaking the integrity check. The cookie is then accepted as valid by the web page.

The next best idea was to add a secret key to what we hash. This way, the user cannot recompute the digest as the secret key is required, much like a MAC. On receipt of the tampered cookie, the page computes `SHA-256(key || tampered_cookie)`, where

|| represents the concatenation of the two values and obtains something that won't match what the malicious user probably sent. Figure 3.10 illustrates this approach.

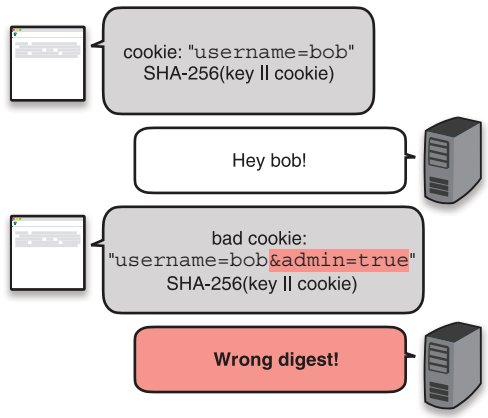


Figure 3.10 By using a key when computing the hash of the cookie, one could think that a malicious user who wants to tamper with their own cookie wouldn't be able to compute the correct digest over the new cookie. We will see later that this is not true for SHA-256.

Unfortunately, SHA-2 has an annoying peculiarity: from a digest over an input, one can compute the digest of an input and more. What does this mean? Let's take a look at figure 3.11, where one uses SHA-256 as `SHA-256(secret || input1)`.

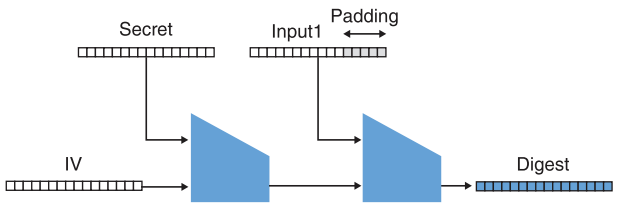


Figure 3.11 SHA-256 hashes a secret concatenated with a cookie (here named `input1`). Remember that SHA-256 works by using the Merkle–Damgård construction to iteratively call a compression function over blocks of the input, starting from an initialization vector (IV).

Figure 3.11 is highly simplified but imagine that `input1` is the string `user=bob`. Notice that the digest obtained is effectively the full intermediate state of the hash function at this point. Nothing prevents one from pretending that the `padding` section is part of the input, continuing the Merkle–Damgård dance. In figure 3.12, we illustrate this attack, where one would take the digest and compute the hash of `input1 || padding || input2`. In our example, `input2` is `&admin=true`.

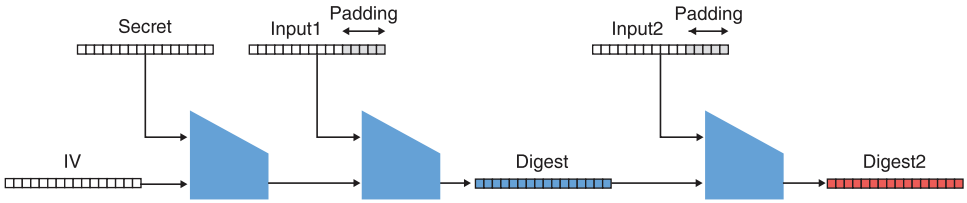


Figure 3.12 The output of the SHA-256 hash of a cookie (the middle digest) is used to extend the hash to more data, creating a hash (the right digest) of the secret concatenated with `input1`, the first padding bytes, and `input2`.

This vulnerability allows one to continue hashing from a given digest, like the operation was not finished. This breaks our previous protocol, as figure 3.13 illustrates.

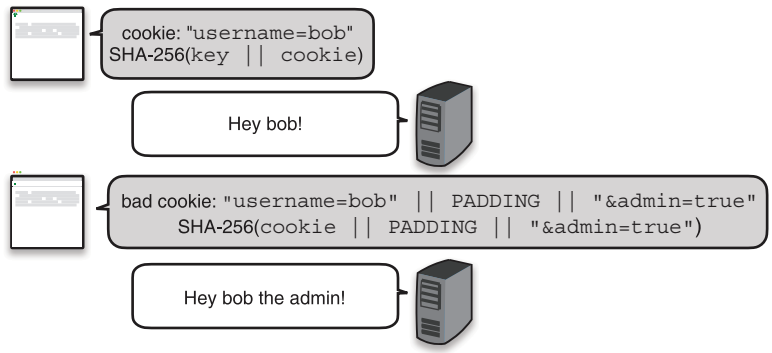


Figure 3.13 An attacker successfully uses a length-extension attack to tamper with their cookie and computes the correct hash using the previous hash.

The fact that the first padding now needs to be part of the input might prevent some protocols from being exploitable. Still, the smallest amount of change can reintroduce a vulnerability. For this reason one should *never* hash secrets with SHA-2. Of course, there are several other ways to do it correctly (for example, `SHA-256(k || message || k)` works), which is what HMAC provides. Thus, use HMAC if you want to use SHA-2 and use KMAC if you prefer SHA-3.

Summary

- Message authentication codes (MACs) are symmetric cryptographic algorithms that allow one or more parties who share the same key to verify the integrity and authenticity of messages.
  - To verify the authenticity of a message and its associated authentication tag, one can recompute the authentication tag of the message and a secret key,

and then match the two authentication tags. If they differ, the message has been tampered with.

  - Always compare a received authentication tag with a computed one in constant time.

- While MACs protect the integrity of messages by default, they do not detect when messages are replayed.
- Standardized and well-accepted MACs are the HMAC and the KMAC standards.
- One can use HMAC with different hash functions. In practice, HMAC is often used with the SHA-2 hash function.
- Authentication tags should be of a minimum length of 128 bits to prevent collisions and forgery of authentication tags.
- Never use SHA-256 directly to build a MAC as it can be done incorrectly. Always use a function like HMAC to do this.